Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 01-017

Parallel Tree Projection Algorithm for Se-quence Mining

Valerie Guralnik, Nivea Garg, and George Karypis

March 29, 2001

Parallel Tree Projection Algorithm for Sequence Mining *

Valerie Guralnik, Nivea Garg, George Karypis
{guralnik, garg, karypis}@cs.umn.edu
Department of Computer Science and Engineering/Army HPCC Research Center
University of Minnesota

February 13, 2001

Abstract

Discovery of sequential patterns is becoming increasingly useful and essential in many scientific and commercial domains. Enormous sizes of available datasets and possibly large number of mined patterns demand efficient and scalable algorithms. In this paper we present two parallel formulations of a serial sequential pattern discovery algorithm based on tree projection that are well suited for distributed memory parallel computers. Our experimental evaluation on a 32 processor IBM SP show that these algorithms are capable of achieving good speedups, substantially reducing the amount of the required work to find sequential patterns in large databases.

1 Introduction

Sequence data arises naturally in many applications. For example, marketing and sales data collected over a period of time provide sequences that can be analyzed and used for projections and forecasting. In the past several years there has been an increased interest in using data mining techniques to extract interesting sequential patterns from temporal sequences. The most time consuming operation in the discovery process of such patterns is the computation of the frequency of the occurrences of interesting sub-sequences of set of events (called candidate patterns) in the database of sequences. However, the number of sequential patterns grows exponentially and various formulations have been developed [AS96, MTV95, SA96, JKK99] that try to contain the complexity by imposing various temporal constraints, and by consider only those candidates that have a user specified minimum support. Even with these constraints, the task of finding all sequential patterns requires a lot of computational resources (*i.e.*, time and memory), making it an ideal candidate for parallel processing.

The algorithms that discover sequential associations are mainly motivated by those developed for non-sequential associations. In particular, there are two main classes of sequential association rule discovery algorithms. The first class of algorithms [SA96, JKK99] was developed by extending the *Apriori* [AS96, SA96] algorithm, and the second class [Zak98, HPMA+00] was developed by extending the tree-projection algorithm [AAP00]. Even though, sequential association rule discovery algorithms based on tree-projection have be shown to substantially outperform those based on *Apriori*, they still require substantial amounts of computational resources. This was recognized by Zaki [Zak99] which developed a parallel formulation of the SPADE algorithm [Zak98] for shared-memory parallel computers. However, there has been no work in developing scalable and efficient parallel formulations for this class of algorithms that are suitable for distributed memory parallel computers.

In this paper we present two different parallel algorithms for finding sequential association rules on distributed-memory parallel computers. The first algorithm decomposes the computation by exploiting data parallelism, whereas the other utilizes task parallelism. One of the key contributions of this paper is the development of a static task decomposition scheme that uses a bipartite graph partitioning algorithm to

This work was supported by NSF CCR-9972519, EIA-9986042, ACI-9982274, by Army Research Office contract DA/DAAG55-98-1-0441, by the DOE ASCI program, and by Army High Performance Computing Research Center contract number DAAH04-95-C-0008. Access to computing facilities was provided by the Minnesota Supercomputing Institute.

simultaneously balance the computations and at the same time reduce the data sharing overheads, by minimizing the portions of the database that needs to be shared by different processors. We experimentally evaluate the performance of our proposed algorithms on different datasets on a 32-processor IBM SP2 parallel computer. Our experiments show that the proposed algorithms incur small communication overheads, achieve good speedups, and can effective utilize the different processors.

The rest of this paper is organized as follows. Section 2 provides some definitions about sequence mining. Section 3 describes the serial tree projection algorithm for sequential association rule discovery. Section 4 describes our different parallel algorithms, and they are experimentally evaluated in Section 5. Finally, Section 6 provide some concluding remarks.

2 Sequence Mining

The problem of mining for sequential patterns was first introduced by Agrawal et al [AS96]. The authors showed how their association rule algorithm for unordered data [AS94] could be adapted to mine for frequent sequential patterns in sequence data. The class of episodes being mined was generalized, and the performance enhancements were presented in [SA96].In this section we will summarize the terminology first introduced by Agrawal [AS96] and being used throughout the paper.

We are given a database $\mathcal{D}$ of sequences called *data-sequences*. Each data-sequence consists of the list of *transactions*, ordered by increasing transaction-time. A transaction has the following fields: sequence-id, transaction-id, transaction-time, and the items present in the transaction. We assume that the set of items $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$, is the set of literals that can be sorted in lexicographical order. The items in the transaction are sorted in lexicographical order.

An **itemset** i is a non-empty set of items, denoted by $(i_1 i_2 \ldots i_m)$, where i_j is an item. The **support of an itemset** is defined as the fraction of the total transactions that contain this itemset. An itemset is said to **frequent** if its support is above a certain user-specified minimum threshold. Given a database of $\mathcal{D}$ of transactions, the problem of mining for association is to find the *all frequent itemsets* among all transactions.

A **sequence** is an ordered list of itemsets, denoted by $< s_1 s_2 \ldots s_n >$, where s_i is an itemset. The **support of a sequence** is defined as the fraction of total data-sequences that contain this sequence. A sequence is said to be **frequent** if its support is above a certain user-specified minimum threshold. Given a database $\mathcal{D}$ of data-sequences, the problem of mining for sequential patterns is to find the *all frequent sequences* among all data-sequences. Each such frequent sequence represents a *sequential pattern*. It is important to note that from now on the term sequential is adjective of pattern, while term serial is adjective of algorithm.

3 Serial Tree Projection Algorithm

Since sequential patterns are essentially associations over temporal data, the algorithm utilizes some of the ideas initially proposed for discovery of associations. The algorithm presented in this paper is based on tree projection algorithm for association rules [AAP00]. The goal of this algorithm is to find all frequent itemsets in the database of transactions. The tree projection algorithm represents discovered itemsets in a lexicographic tree structure. We assume that a lexicographical ordering exists among the items in the database.In this tree, called *Projection Tree*, each node is associated with a k-itemset. A node can be extended into multiple children nodes via items that are lexicographically larger than all the items in its itemset. The new children represent $(k+1)$-itemset and are called their parent's extensions.

For example, let's assume that 1-itemset (1), (2) and (3) are frequent. An item 3 can extend itemset (1) and (2) by creating itemset (1 3) and (1 3), but item 2 can only extend itemset (1) by creating itemset (1 2). An extension (3 2) cannot be created since item 3 is lexicographically larger than item 2.

Tree projection algorithm grows the tree progressively such that only the nodes corresponding to frequent itemsets are generated. The level-wise version of the algorithm grows the tree in a breadth-first manner. In iteration k, it extends all the nodes at level $k-1$. The candidate extensions of a given node are formed by using only the frequent extensions of its parent. All the nodes that belong to a sub-tree which potentially can be extended are called *active extensions*. If the node cannot be extended any further it becomes *inactive* and is pruned from the tree.

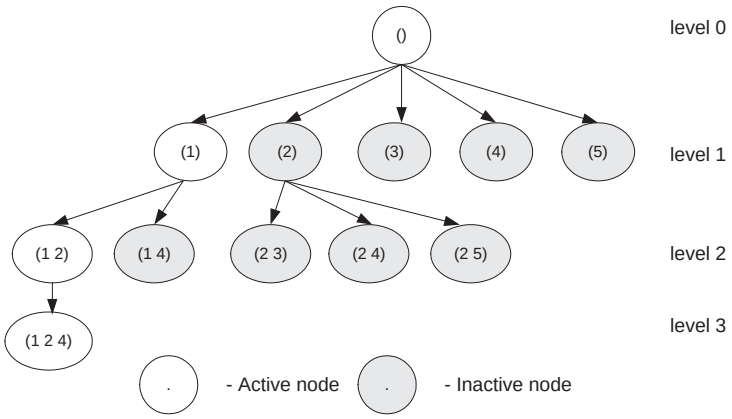

Figure 1: An example of Projection Tree for Associations

Figure 1 illustrates an example of Projection Tree that has just been expanded to level 3. In this example the only active nodes are (), (1), (1 2) and (1 2 4). The other nodes could not be extended any further and therefore became inactive.

One of the key features of the algorithm is that the support of the itemsets represented by the candidate extensions are gathered by using the set of *projected* transactions at the parent. Let's briefly describe the concept of projection. The algorithm maintains a list of *active items* of the node. Active item list of a node consists of items that can be found in itemset represented by its descendants. When a transaction is projected on a node, only the items that occur in its active item list are kept. The transaction gets recursively projected along the paths determined by active extensions. The idea is, only those items in a transaction percolate down the tree that can only potentially be useful in extending the tree by one more level. With every pass of the algorithm, many extensions become progressively inactive, which in turn results in the reduction of active item list sizes at all the nodes. Thus, the size of the projected transaction set reduces progressively. This yields the algorithm its efficiency in the counting phase.

The process of counting support of $(k+1)$-itemsets is accomplished in the following way. Each node, representing the itemset $I = (i_1 i_2 \ldots i_{k-1})$, at level $k-1$ maintains a count matrix, which is used to count support of candidate extensions $(i_1 i_2 \ldots i_{k-1} i_k i_{k+1})$, where items i_k and i_{k+1} are active extensions of the node. Once transactions are projected to the node with matrix, the algorithm iterates through them to gather count of the matrix.

The Tree Projection Algorithm for discovering frequent sequential pattern borrows most of its ideas from Tree Projection algorithm for discovering associations. Thus, the discovered sequential patterns are arranged in a tree data structure, where each node is associated with k-item sequential pattern. However, in a sequential tree projection algorithm a node can be extended into multiple children nodes via items in two ways as follows. A child of the node can be created by extending the last itemset in a pattern represented by a node with an item that is lexicographically larger than all the items in that itemset (called *itemset extension*) or by extending a pattern with a new 1-item itemset (called *sequence extension*). The new children represent $(k+1)$-item patterns.

For example, let's assume that 1-item patterns $< (1) >$, $< (2) >$ and $< (3) >$ are frequent. An item 3 can extend pattern $< (2) >$ by creating both an itemset extension $< (2\ 3) >$ and sequence extension $< (2)\ (3) >$. However an item 1 can extend pattern $< (2) >$ only be creating a sequence extension $< (2)\ (1) >$. An itemset extension $< (2\ 1) >$ cannot be created since item 2 is lexicographically larger then item 1.

Figure 2 illustrates an example of Projection Tree. In this example the set of active extensions of node $< (2) >$ is $\{1,\ 3\}$, where 1 is an active sequence extension and 3 is an active itemset extension. The set of active items is $\{1,\ 2,\ 3\}$.

Another difference between to algorithms is how the counting support of $(k+1)$-item pattern is accomplished. Each node, representing the pattern $P =< s_1 s_2 \ldots s_m >$, at level $k-1$ maintains four count matrices. The first matrix is a lower-triangular matrix, which is used to count support of candidate extensions $< s_1 s_2 \ldots (s_m\ i\ j) >$, where items i and j are active itemset extensions of the node. The second

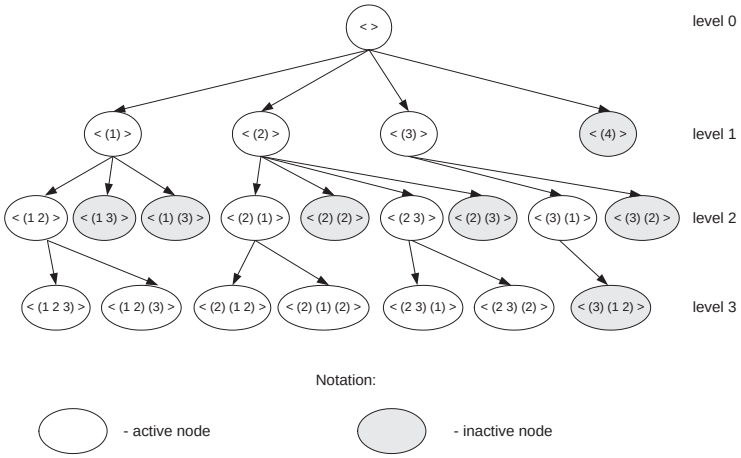

Figure 2: An example of Projection Tree

matrix is used to count support of candidate extensions $< s_1 s_2 \ldots s_m\ (i)\ (j) >$, where item i is an active itemset extension of the node and item j is an active sequence extension of the node. The third matrix is a lower-triangular matrix, which is used to count support of candidate extensions $< s_1 s_2 \ldots s_m\ (i\ j) >$, where items i and j are active sequence extensions of the node such that item j is lexicographically larger then item item i. The last matrix is used to count support of candidate extensions $< s_1 s_2 \ldots (s_m\ i)\ (j) >$, where item i is an active itemset extension of the node and item j is an active sequence extension of the node. Once sequences are projected to the node with matrices, the algorithm iterates through them to gather counts of matrices.

4 Parallel Formulation

The overall structure of the computations performed by the serial tree projection algorithm for discovering sequential patterns (discussed in Section 3)) is encapsulated in the tree that is generated by the algorithm. In particular, if we use the breadth-first approach for tree expansion, the computation proceeds as follows. Initially, there is a single node (*i.e.*, the NULL node), which is expanded to generate all sequential patterns of length one, leading to a tree of depth one. Then, the nodes of that tree that correspond to sequential patterns whose support satisfy the given minimum support constraints are expanded to generate sequential patterns of length two, leading to a tree of depth two. This process of level-by-level tree expansion continues until the tree cannot be grown any further because none of the leaf nodes (*i.e.*, candidate sequential patterns) satisfy the minimum support constraints.

The bulk of the computation in the tree projection algorithm is performed in determining which of the nodes (*i.e.*, sequential patterns) of the tree satisfy the minimum support constraints. Again, as discussed in Section 3, this is done by projecting the original sequences into each of the nodes of the tree and then counting the support using the various matrices. It is important to emphasize, that when counting the support of the nodes at level k, the database sequences are projected to the parent nodes at level $k-2$. As discussed in [AAP00], this is done solely for performance reasons.

Given that the computations are structured in this fashion, there are two general ways that can be used to decompose the computations [KGGK94]. The first approach exploits the data parallelism that exists in computing the support at each node, whereas the second approach exploits the task parallelism that exists in the tree-based nature of the computation. Our parallel formulations using both of these approaches are described in the rest of this section.

4.1 Data Parallel Formulation

The key idea behind our data parallel formulation (DPF) is to decompose the computations associated with counting the support of the various sequential patterns at each node of the tree. In particular, our parallel

formulation works as follows.

If p is the total number of processors, the original database is initially partitioned into p equal size parts, and each one is assigned to a different processor. To compute the support of the candidate sequences at level k, each processor projects its local set of data sequences to the nodes at level $k-2$, and computes their support based on the local data sequences. The global supports are determined by using a reduction operation to add up the individual supports. These global supports are made known to all the processors, and are used to determine which nodes at the kth level meet the minimum support constraints. Note that in this approach, all the processors build the same tree (which is identical to that built by the serial algorithm). This parallel formulation is similar in nature to the count distribution method developed for parallelizing the serial Apriori algorithm for finding associative patterns [AS96, HKK99].

Since the database is equally distributed among the processors, the overall computation is well balanced[1]. However, if m_k is the number of candidate sequential patterns at level k, the communication overhead of this formulation due to the reduction operation is $O(m_k)$ (since $m_k >> p$, [KGGK94]). Let t_k^s be the amount of time required by the serial algorithm to compute the support of the candidate patterns at level k. The proposed parallel algorithm will perform the same step in the following time:

$$t_k^p = t_k^s/p + O(m_k). \tag{1}$$

If n is the total number of sequences in the database, and f_{k-2} is the number of nodes at level $k-2$ of the tree, then

$$t_k^s = O(n * f_{k-2}), \tag{2}$$

since each sequence needs to be projected to each one of the nodes at level $k-2$. Combining Equations 2 and 1, we can see that the maximum number of processors that can be used cost-effectively is given by

$$p = O\left(\frac{n * f_{k-2}}{m_k}\right).$$

This formula indicates that DPF algorithm leads to scalable formulations only when the overall work increases as a result of an increase in the database size. In this case, as n increases, we can use more processors, and still achieve good parallel efficiency. However, if the overall work increases as a result of a decrease in the minimum support, in that case n stays the same, but both m_k and f_{k-2} increase, at about the same rate. As a result, we cannot use more processors as the overall work increases, without a decrease in the parallel efficiency.

Furthermore, this algorithm works well only when the tree and count matrices can fit into the main memory of each processor. If the number of candidates is large, then the matrices may not fit into the main memory. In this case, this algorithm has to partition the tree and compute the counts by scanning the database multiple times, once for each partition of the tree.

4.2 Task Parallel Formulation

The key idea behind the task parallel formulation (TPF) is that when the support of the candidate patterns at level k is computed by projecting the databases at the various nodes at the $k-2$ level of the tree, the computations at each of these $k-2$ nodes are independent of each other. Thus, the computations at each node becomes an independent task and the overall computation can be parallelized by distributing these tasks among the available processors.

Our task parallel formulation distributes the tasks among the processors in the following way. First, the tree is expanded using the data-parallel algorithm described in Section 4.1, up to a certain level $k+1$, with $k > 0$. Then, the different nodes at level k are distributed among the processors. Once this initial distribution is done, each processor proceeds to generate the subtrees (*i.e.*, sub-forest) underneath the nodes that it has been assigned.

In order for each processor to proceed independently of the rest, it must have access to the sequences in the database that may contain the patterns corresponding to the nodes of the tree that it has been assigned.

[1] This can be ensured by partitioning the database in a random fashion, so that no processor will be assigned a portion of the database that has more frequent patterns than others.

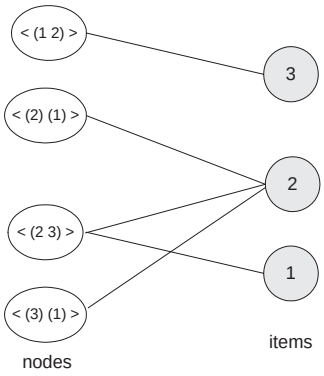

Figure 3: A bipartite graph corresponding to projection tree

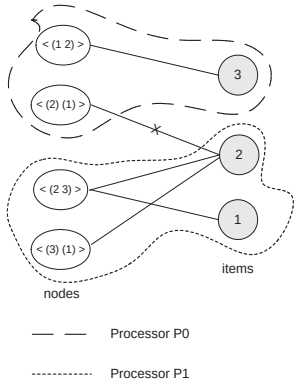

Figure 4: A partitioned bipartite graph

The database sequences (or portions off) that each processor P_i needs to have access to, can be determined as follows. Let S_i be the set of nodes assigned to P_i, A_i be the union of all the active items of the nodes in S_i, and B_i be the union of all the items in each of the frequent patterns that correspond to a node in S_i. Given these definitions, each processor needs to have access to all the sequences that contain items belonging in the set $C_i = A_i \bigcup B_i$. Moreover, since it computes only the frequent patterns that are underneath the nodes in S_i, it needs to only retain the items from the original sequences that are in C_i. We will refer to the set of items C_i as the *sub-forest itemset* of the node set S_i.

In our algorithm, once the distribution of the nodes at level k is determined, the sets $C_0, C_1, \ldots, C_{p-1}$ are determined and broadcasted to all the processors. Each processor then reads the local portion of the database (the one that was used by the data-parallel algorithm), splits it into p parts, one for each processor, and sends it to the appropriate processor. Each processor, upon receiving the sequences corresponding to its portion of the tree, writes them to the disk and proceeds to expand its nodes independently. Note that since the sets C_i for different processors can overlap, processors will end up having overlapping sections of the original database.

The key step in the STPF algorithm is the method used to partition the nodes of the kth level of the tree into p disjoint sets $S_0, S_1, \ldots, S_{p-1}$. In order to ensure load balance, this partitioning must be done in a way so that the work is equally divided among the different processors. A simple way of achieving this is to assign a weight to each node based on the amount of work required to expand that node, and then use a bin-packing algorithm [CLR90] to partition the nodes into p equal-weight buckets. This weight can be either a measure of the actual computational time that is required, or it can be a measure that represents a relative time, in relation to the time required by other nodes. Obtaining relative estimates is much easier than obtaining estimates of the actual execution time. Nevertheless, accurately estimating the relative amount of work associated with each node is critical for the overall success of this load balancing scheme.

A simple estimate of the relative amount of work of a particular node is to use the support of its corresponding sequential pattern. The motivation behind this approach is that if a node has a high support it will most likely generate a deeper subtree, than a node with a lower support. However, this estimate is based on a single measure and can potentially by very inaccurate. A better estimate of the relative amount of work can be obtained by summing up the support of all of its active extensions, that is, the support of all of its children nodes at the $k+1$st level in the tree. This measure by looking ahead at the support of the patterns of length $k+1$, will lead to a more accurate estimate. This is the method that we use to estimate the relative amount of work associated with each node.

Even though this bin-packing-based approach is able to load-balance the computations, it may lead to partitions in which the sub-forest itemsets assigned to each processor have a high degree of overlap. Consequently, the follow-up database partitioning will also lead to highly overlapping local databases. This increases the amount of time required to perform the partitioning, the amount of disk-storage required at each processor, and as we will see in the experiments presented in Section 5.2, it also increases the amount of time required to perform the projection. Ideally, we will like to partition the nodes at the kth level in such

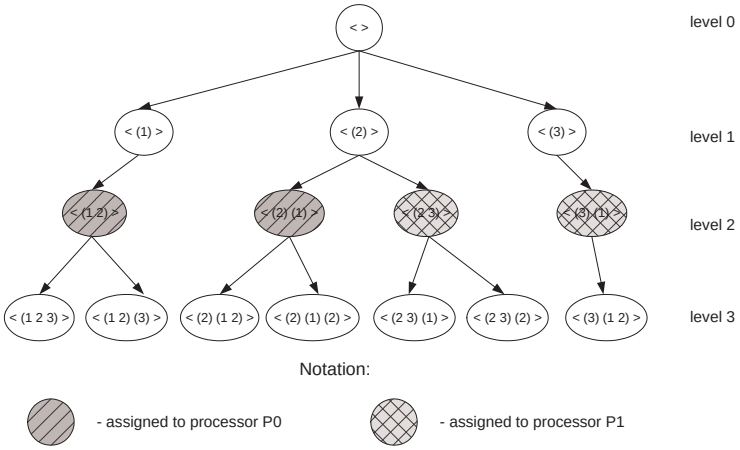

Figure 5: A tree with nodes assigned to two processors

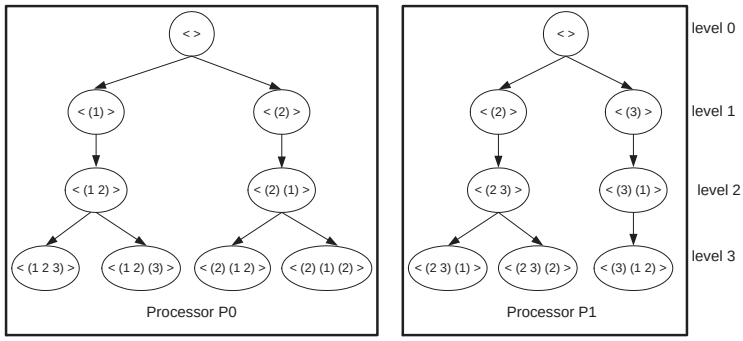

Figure 6: Projection Trees distributed to two processors

a way so that in addition to balancing the load we also minimize the degree of overlap among the different databases assigned to each processor.

Since the degree of overlap in the local databases is directly proportional to the degree of overlap in the sub-forest itemsets, we can minimize the database overlap by minimizing the overlap in the sub-forest itemsets. This later problem can be solved by using a minimum-cut bipartite graph partitioning algorithm, as follows.

Let $G = (V_A, V_B, E)$ be an undirected bipartite graph, where V_A, and V_B are the two sets of vertices and E is the set of edges. The vertices in V_A correspond to the nodes of the tree, the vertices in V_B correspond to the sequence items, and there is an edge $(u, v) \in E$ with $u \in V_A$ and $v \in V_B$, if the item b is an active item of node u. Each vertex $u \in V_A$ has a weight $w(u)$ that is equal to the relative amount of work required to expand its corresponding subtree, and each vertex $v \in V_B$ has a weight of one. Furthermore, each edge (u, v) has a weight of one.

A partitioning of this bipartite graph into p parts that minimizes the edge-cut (*i.e.*, the number of edges that straddle partitions), subject to the constraint that the the sum of the weight of the vertices in V_A assigned to each part is roughly the same, can be used to achieve the desired partitioning of the tasks. Since each partition contains tree-nodes whose total estimated work is roughly the same, the overall computation will be balanced. Furthermore, by minimizing the edge-cut the resulting partition groups nodes together so that their sub-forest itemsets have as little overlap as possible. Note that an edge belonging to the cut-set indicates that the corresponding item is shared between at least two partitions. In general, for each item u, the number of its incident edges that belong to the cut-set plus one, represent the total number of sub-forest itemsets that this node belongs to.

Let's consider an example of the tree $\mathcal{T}$ illustrated in Figure 2. This tree is expanded to level 3, assuming that two processors are available we want partition T based on nodes at level 2 in two partitions. The graph

Dataset	Avg. no. of transactions per sequence	Avg. no. of items per transaction	Avg. length of maximal potentially frequent sequences	Avg. size of itemsets maximal potentially frequent sequences	$MinSup(\%)$
(1)	10	2.5	4	1.25	0.1
(2)	10	5	4	1.25	0.25
(3)	10	5	4	2.5	0.33
(4)	20	2.5	4	1.25	0.25

Table 1: Parameter values for datasets

N.of P	DPF	TPF-GP2	TPF-GP3	TPF-BP2	TPF-BP3
2	7914.82s	4370.37s	4263.24s	6128.41s	6885.33s
4	7922.66s	3012.61s	3443.54s	4835.05s	6740.49s
8	8017.73s	2346.55s	2785.41s	3626.02s	6294.13s
16	8135.03s	1992.7s	2595.51s	2903.85s	5508.26s
32	8413.89s	1804.09s	2497.52s	2404.73s	4604.7s

Table 2: Serial execution time for $C10 - T5 - S4 - I2.5$

G corresponding to tree T is illustrated in Figure 3. Let's assume that graph G was partitioned as shown in Figure 4. This partition results in assigning nodes $<$ (1 2) $>$ and $<$ (2) (1) $>$ to processor P_0; nodes $<$ (2 3) $>$ and $<$ (3) (1) $>$ are assigned to processor P_1 (see Figure 5). Each processors is then distributed a new tree as illustrated in Figure 6.

In our algorithm, we compute the bipartite min-cut partitioning algorithm using the multi-constraint graph partitioning algorithm [KK98] available in the METIS graph partitioning package [KK95].

5 Experimental Evaluation

5.1 Experimental Setup

We use the same synthetic datasets as in [SA96], albeit with more data-sequences. We generated datasets by setting number of maximally potentially frequent sequences $N_S = 5000$, number of maximal potentially frequent itemsets $N_I = 25000$ and number of items. $N = 10000$. The number of data-sequences D was set to 1 million. Table 1 summarizes the dataset parameter settings and shows the minimum support used in experiments reported below.

We ran our experiments on IBM SP cluster. The IBM SP consists of 79 four-processor and 3 two-processor machines with 391 GB of memory. The machines utilize the 222 MHz Power3 processors.

5.2 Results

We evaluated DPF and TPF based on the following criteria: how effective they are in terms of execution time and how effective they are in balancing workload. Additionally, we implemented two approaches for balancing the workload in TPF. One was based on Bipartite Graph Partitioning (TPF-GP) and the other was based in Bin Packing (TPF-BP). Those schemes were also evaluated based on how well they can minimize the overlap between databases assigned to each processor. Both schemes were run starting the task parallelism at level 2 and level 3 and resulting in 4 sets of experiments for each of the datasets (TPF-GP2, TPF-GP3, TPF-BP2, TPF-BP3).

Figure 7, Figure 8, Figure 9 and Figure 10 show the execution time of all five schemes on the four different datasets on 2,4, 8, 16 and 32 processors. A number of interesting observations can be made looking at these results.

First, the DPF algorithm achieves good speedups for all four data sets. In particular, as we increase the number of processors from 2 to 32 (a factor of 16), the amount of time decreases by a factor of 12.3, 13.9, 13.42 and 13.64 for each one of the four datasets, respectively.

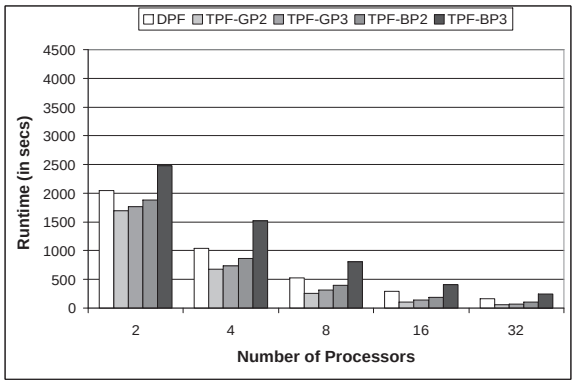

Figure 7: (1)

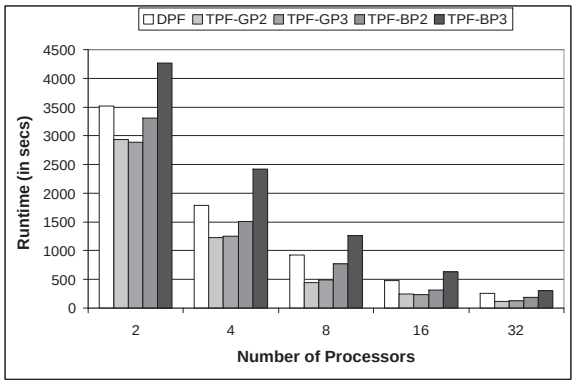

Figure 8: (2)

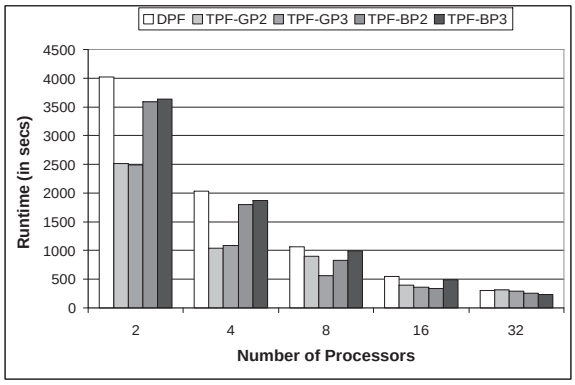

Figure 9: (3)

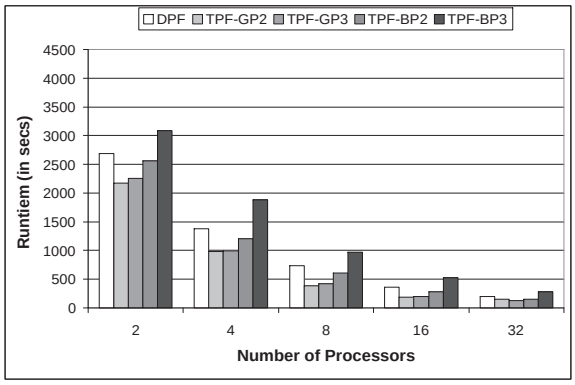

Figure 10: (4)

Second, the speedups achieved by the TPF-GP2 algorithm are actually sometimes super-linear. As the number of processors increases from 2 to 32, the runtime for each one of the four datasets decreases by factor of 29.8, 24.5, 8.0 and 14.66, respectively. The super-linear speedup is due to the fact that each processor is assigned a sub-forest of the original tree and the databases are re-distributed. As a result, the amount of time spent in projection and disk I/O actually reduces as the number of processors increases. Also the poor result on the third data set is due to the fact that the static tasks assignment leads to unbalanced work distribution.

Third, comparing the bipartite graph partitioning based formulation of TPF to the one based on bin-packing we can see that graph partitioning leads to substantially smaller execution times. This is because of the two reasons. First, as discusses in Section 4.2, the graph partitioning based approach reduces the overlap among the local databases; thus reducing the redistribution cost as well as disk I/O. Second, because each processor is assigned fewer distinct items, the projection cost is reduced. To illustrate the reduction in database sizes we plotted the size of local databases (summed over all the processors) for the two sets of schemes. These results are shown in Figure 11, Figure 12, Figure 13 and Figure 14. These figures show the database sizes, relative to the size of the databases (after it has been pruned so it only contains the active items) on a single processor. As we can see from those figures, the graph partitioning based schemes lead to local databases whose size up to a factor smaller that the bin-packing scheme.

Also, the reduction in projection time can be seen in Table 2, that shows the total amount of time spent by all the processors in performing the local computation. From this table we can see that the graph partitioning scheme TPF-GP2 spends only 1804 secs versus 2404 secs for TPF-BP2.

Fourth, comparing TPF-GP2 versus TPF-GP3 and TPF-BP2 against TPF-BP3, we can see that the run-times achieved by the schemes that switch to tasks distribution after the second level are in general smaller. The only exception is for the third dataset in which TPF-GP3 does better that TPF-GP2 on 8, 16 and 32 processors. This is due to the fact that TPF-GP3 lead to better load balance and that TPF-GP2 is

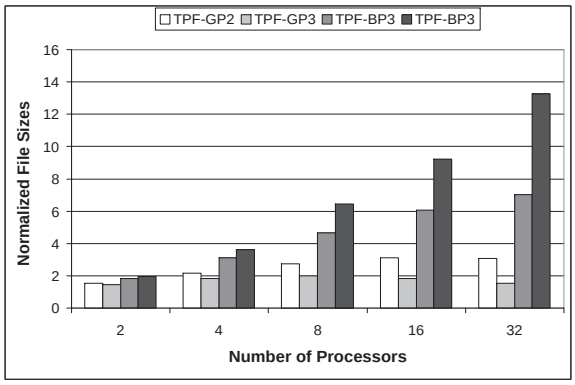

Figure 11: (1)

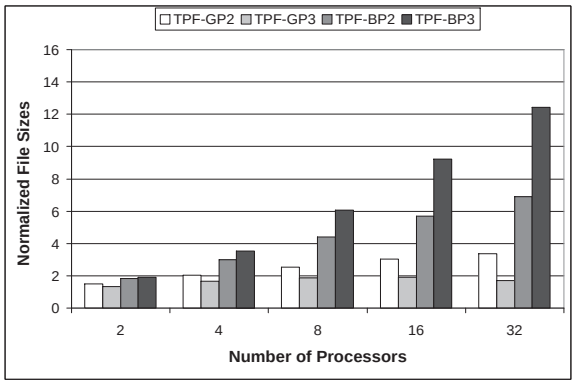

Figure 12: (2)

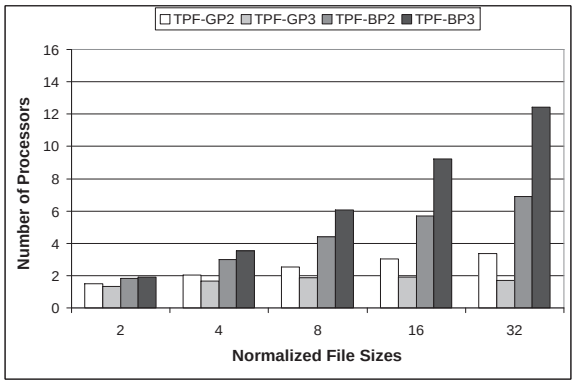

Figure 13: (3)

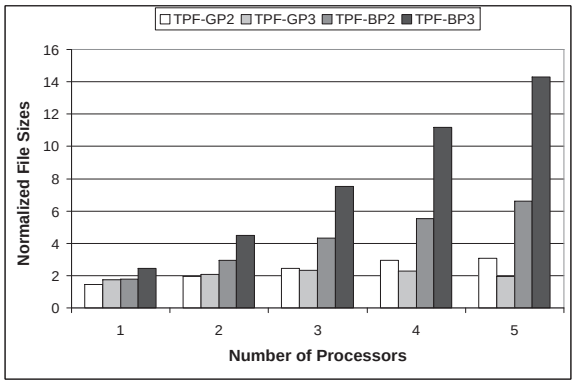

Figure 14: (4)

highly unbalanced.

6 Conclusion and Directions of Future Research

In this paper we presented two algorithms for finding sequential patterns using the tree projection algorithm that are suitable for distributed memory parallel computers. Our experimental results show that both the data parallel and the task parallel formulations are able to achieve good speedups as the number of processors increase. Furthermore, the bipartite graph partitioning based task distribution approach is able to substantially reduce the overlap in the databases required by each processor.

Despite these promising results we believe that the overall performance can be further improved by developing dynamic load balancing schemes. One way of doing this is to modify the TPF algorithm so that instead of expanding its different subtrees asynchronously, to do so in a level by level fashion, and check to see if the work needs to be redistributed after each level. Our preliminary experiments with such an approach appear promising and is the focus of our current research.

References

[AAP00] R.C. Agarwall, C. Aggarwal, and V.V.V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, Santiago, Chile, 1994.

[AS96] R. Aggrawal and R. Srikant. Mining sequential patterns. In *Proc. of the Int'l Conference on Data Engineering (ICDE)*, Taipei, Taiwan, 1996.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, New York, NY, 1990.

[HKK99] E.H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Eng. (accepted for publication)*, 1999.

[HPMA+00] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.C. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *Proc. 2000 Intl. Conference on KDD*, 2000.

[JKK99] Mahesh V. Joshi, George Karypis, and Vipin Kumar. Universal formulation of sequential patterns. Technical report, Universit of Minnesota, Department of Computer Science, Minneapolis, 1999.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[KK95] G. Karypis and V. Kumar. metis: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1995. Available on the WWW at URL *http://www.cs.umn.edu/~karypis/metis/metis.html*.

[KK98] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of Supercomputing*, 1998. Also available on WWW at URL http://www.cs.umn.edu/~karypis.

[MTV95] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. of the First Int'l Conference on Knowledge Discovery and Data Mining*, pages 210–215, Montreal, Quebec, 1995.

[SA96] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.

[Zak98] M.J. Zaki. Efficient enumeration of frequent sequences. In *7th International Conference on Information and Knowledge Management*, 1998.

[Zak99] Mohammed J. Zaki. Parallel sequence mining on smp machines. In *Workshop On Large-Scale Parallel KDD Systems (in conjunction 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 57–65, San Diego, CA, august 1999.